2006

Ordered Pattern Matching: Towards Full-Text Retrieval

Wing-Kail Hon

Rahul Shah

Jeffrey S. Vitter
Kansas University, jsv@ku.edu

Report Number:
06-008

ORDERED PATTERN MATCHING: TOWARDS FULL-TEXT RETRIEVAL

Wing-Kai Hon
Rahul Shah
Jeffrey Scott Vitter

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #06-008
March 21, 2006

Ordered Pattern Matching: Towards Full-Text Retrieval

Wing-Kai Hon Rahul Shah Jeffrey Scott Vitter*

Abstract

A typical query in Information Retrieval consists of multiple keywords, and the target is to retrieve matching documents. Various selection heuristics and ranking criteria are based on the proximity of keywords. For this purpose, it helps if the positions of the keywords in the document are listed in sorted order. For traditional documents consisting of lists of words, existing data structures like inverted indexes serve this purpose well. In this paper, we extend the study to full-text searching where each document is a single string of characters; this problem is well motivated in biological sequence mining.

A key building block for such a functionality is a set of queries which can perform pattern matching with constraints on text positions. Let $T[0..n-1]$ be the text. We preprocess this text and build a data structure to answer the following queries efficiently: (1) Given a pattern P, a position p and a rank k, output the position of the kth (in text order) occurrence of P in $T[p..n-1]$, and (2) Given a pattern P, and positions i and j, report (or count) all occurrences of P in $T[i..j]$. Then, we show how to use these queries to solve various problems including document retrieval, proximity searching and aligned pattern matching.

One point to notice is that: Each of the above queries can also be considered as a two-dimensional range query, and there are existing geometric data structures with good query performance. Nevertheless, the design of our data structure follows an alternative approach, which is based on augmenting a binary search tree with bit dictionaries.

1 Introduction

In traditional information retrieval [4], the text is broken into keywords and then queries are based on these keywords. The most popular data structure to efficiently address these queries is the inverted index, which stores for each keyword, the list of page numbers where the keyword appears. When there are multiple keywords in a query, the list of pages corresponding to each keyword is retrieved, then the intersection of these sets (corresponding to each keyword) is taken, and finally the

*Department of Computer Sciences, Purdue University, West Lafayette, IN 47907–2066, USA. {wkhon,rahul,jsv}@cs.purdue.edu

pages are ranked. The traditional *tfidf* ranking takes the frequencies of keyword terms into account for ranking the pages. Sometimes, the inverted index stores for each keyword not only the list of page numbers where the keyword appears, but also the positions of the occurrence of the keyword within a page. This augmentation allows scores of new functionalities and ranking heuristics. One such example is the locality-based similarity heuristic [7].

In full-text retrieval systems, the text cannot be broken into keywords. Basically, the text is a string of characters and the query pattern can match starting from any position in the text. When a query consists of a single keyword (pattern), data structures based on suffix trees [20, 18, 10] and suffix arrays [17] can list all matches of the pattern efficiently. Consider a query which consists of two patterns P_1 and P_2, and we wish to list all matches of P_1 and P_2 which are in some sense "near" to each other. There are approximate pattern matching data structures which allow limited wildcards [6], but they are not well suited for nearness queries in general. The naive way of using suffix trees is to obtain the lists of all occurrences of P_1 and P_2 and then compute their intersection. However, the problem is that number of occurrences of P_1 or P_2 alone can be significantly larger than the number of desired occurrences. It could be horribly inefficient to check all occurrences of P_1 and P_2 individually.

There are many good algorithms and heuristics known for computing the intersection of two list, when they are in the sorted order and random access to each list is possible [8, 9]. For many input lists, one does not need to check all the entries in the list but only those when the merging of two lists alternates. In the above case, if the number of occurrences of P_2 is much smaller than P_1, an even simpler heuristic is to search for occurrences of P_1 in the vicinity of the list of occurrences outputted by P_2. Such heuristics need some handle on text positions when querying. We consider the following three pattern matching queries with view to augmenting such functionalities. Let $T[0..n-1]$ be the text.

Range_Count(pattern P, position p, position q): Counts the number of occurrences of a pattern P in text T, where the matching location starts between the text positions p and q.

Range_Report(pattern P, position p, position q): The reporting version of the Range_Count query. Reports all occurrences of P in T where the matching location starts between the text positions p and q.

Select(pattern P, position p, rank k): Among all occurrences of P in T with the matching location after text position p, reports the occurrence with the kth smallest matching location.

We show that *Range_Count* and *Select* queries can be answered in $O(|P| + \log n)$ time. For *Range_Report* query, it can be answered $O(|P| + occ \log n)$ time. We also describe an augmentation of our data structure so that it takes $O(|P| + \log n + occ)$ time but using $O(n \log n)$ words of space.

Based on these queries, we can have quick random access to the list of occurrences of the pattern in text by text order. The *Select* query enables us to compute faster intersections using Demaine et al.'s [8] method. We can address a number of other more

complex queries. Consider a database-like proximity query: Select all occurrences i of P_1 in T, where there are k occurrences of P_2 in $T[i-d,...,i+d]$. That is, select only those occurrences of P_1 which have sufficient number of P_2's in its vicinity. This can be addressed well by the *Range_Count* query. Other problems, like the document retrieval problem [19], can also be answered using these queries.

To the best of our knowledge, this is the first time such queries are formally defined and solved as a building block towards more complex full-text queries. This work lies at the heart of the intersection of fields like databases, information retrieval, and full-text indexing. It can be very useful in areas like bioinformatics.

Our data structures are closely related to the ones used for solving two dimensional range searching problems in computational geometry [1, 2, 3, 5]. However, we give a somewhat different approach of using rank-select dictionaries to augment binary search trees. Since dictionaries are elegantly demarcated from the main structure our data structure can be easily externalized. We believe that one could benefit from various dictionary compression methods and hence our data structures might be easier to convert into space compressed versions like compressed suffix arrays [13, 15, 14]. At the least, we match the best space-time bounds of data structures in geometry. We add a new primitive of adding ranked queries into 2-D search.

2 Preliminaries

In this section, we first give some basic notations and define the suffix array of a text T. Then, we define the bit-rank query on a bit-vector and state a data structure result of Jacobson [16] which supports such query in constant time.

2.1 Suffix Array

Let T be a text over an alphabet Σ. We assume that T ends with a special symbol \$ not appearing anywhere else inside the text, and \$ is lexicographically smaller than the other characters in Σ. Let n be the number of characters (including \$) in T. We assume that T is stored in an array $T[0..n-1]$, where $T[n-1] = \$$. For any integer i in $[0, n-1]$, we denote

- $T[i]$ as the $(i+1)$th character of T from the left; and
- T_i as the suffix of T starting from position i; that is, $T_i = T[i..n-1] = T[i]T[i+1]...T[n-1]$.

The *suffix array* of T [17], denoted SA$[0..n-1]$, is a representation for the sorted sequence of the suffixes of T. Formally, SA$[i]$ stores the starting position of the $(i+1)$th smallest suffix of T. In other words, according to the lexicographical order, $T_{\mathrm{SA}[0]} < T_{\mathrm{SA}[1]} < \ldots < T_{\mathrm{SA}[n-1]}$. See Figure 1 for an example. Note that SA$[0] = n-1$. The suffix array can be stored in $O(n \log n)$ bits, and it supports efficient pattern matching based on two simple observations: Firstly, if a pattern P occurs in some location i in T, then P is a prefix of the suffix T_i; secondly, for all suffixes (whose prefix) matching with P, they are in consecutive locations in the suffix array. For

i	T_i
0	acaaccg$
1	caaccg$
2	aaccg$
3	accg$
4	ccg$
5	cg$
6	g$
7	$

i	SA[i]	$T_{\mathrm{SA}[i]}$
0	7	$
1	2	aaccg$
2	0	acaaccg$
3	3	accg$
4	1	caaccg$
5	4	ccg$
6	5	cg$
7	6	g$

Figure 1: The suffixes and the suffix array of $T =$ acaaccg$.

instance, in Figure 1, $SA[4..6]$ corresponds to the suffixes matching with c and $SA[2..3]$ corresponds to suffixes matching with ac.

Given the text $T[0..n-1]$ and its suffix array, Manber and Myers [17] showed that we can find the range (i, j) in the SA that matches a pattern P in $O(|P| \log n)$ time using binary search on SA. They further showed how to improve the time complexity to $O(|P| + \log n)$ time by storing an additional data structure of $O(n \log n)$ bits. (Please refer to the original paper for more details.) After the range is computed, the occurrences of P in T can be reported in $O(j - i + 1)$ time by accessing $SA[i], SA[i+1], \ldots, SA[j]$.

2.2 Data Structure for the Bit-Rank Query

Given a bit-vector $B[0..n-1]$, the $bit\_rank(i, b)$ query reports the number of occurrences of the bit b in $B[0..i]$. The following lemma states a data structure result for supporting these queries in constant time.

Lemma 1 ([16]) *Let B be a bit-vector of n bits. We can construct a data structure of $o(n)$ bits so that together with B, the bit_rank query on B can be supported in constant time.*

3 Ordered Pattern Matching Structure

Before we give the details of our auxiliary data structure for supporting various ordered pattern matching queries, we first define a notation $B_{x,y}$ which will be useful in later discussion. Assume that all SA entries are stored in $\lceil \log n \rceil$ bits, and let us consider those entries whose first x bits form the binary representation of y in x bits. Let $SA[i_1], SA[i_2], SA[i_3], \ldots$ be all such entries, with $i_1 < i_2 < i_3 < \ldots$. Then, $B_{x,y}$ is a bit-vector such that its jth bit $B_{x,y}[j]$ stores the $(x+1)$th most significant bit (MSB) of $SA[i_j]$. We also define the bit-vector $B_{0,0}$ such that its jth bit $B_{0,0}[j]$ stores the first MSB of $SA[j]$. For instance, referring to the suffix array in Figure 1, we have

i	0	1	2	3	4	5	6	7
SA[i]	7	2	0	3	1	4	5	6
first MSB	1	0	0	0	0	1	1	1
second MSB	1	1	0	1	0	0	0	1
third MSB	1	0	0	1	1	0	1	0

- $B_{0,0} = (1, 0, 0, 0, 0, 1, 1, 1)$;
- $B_{1,0} = (1, 0, 1, 0)$, $B_{1,1} = (1, 0, 0, 1)$;
- $B_{2,0} = (0, 1)$, $B_{2,1} = (0, 1)$, $B_{2,2} = (0, 1)$, $B_{2,3} = (1, 0)$.

Our auxiliary data structure is analogous to the wavelet tree in [14]. It is a complete and balanced binary tree with n leaves, with a total of $\lceil \log n \rceil$ levels. Each node or leaf is augmented with a particular bit-vector as follows. The root node is augmented with $B_{0,0}$. For a node at the ith level augmented with $B_{i,j}$, its left child will be augmented with $B_{i+1,2j}$, and its right child will be augmented with $B_{i+1,2j+1}$. In addition, for each such bit-vector B, we construct the corresponding data structure of Lemma 1 to support constant-time *bit_rank* query on B.

Once we have computed the range of SA entries $SA[i..j]$ matching P, we show how the above auxiliary data structure can be used to solve a restricted version of *Select* and *Range_Count* query efficiently. Then, we show how to apply the restricted version to solve the general queries, and to solve *Range_Report* in the desired time bound. Finally, we give the space analysis of the auxiliary data structure.

Select$(P, 0, k)$ Query—the restricted Select: To find the kth occurrence of P in T, say at position p_k, we will traverse the binary tree of our auxiliary data structure from the root downwards such that at the ith node we traverse, we retrieve its ith MSB of p_k. The idea is that when we are visiting a level-x node in the binary tree, we identify the range i_x and j_x within the augmented bit-vector $B_{x,y}$ that stores all candidates of the $(x+1)$th bit of p_k. We also identify the value k_x so that the number of 0's in $B_{x,y}[i_x..j_x]$ is at least k_x *if and only if* the $(x+1)$th bit of p_k is 0. For instance, at the root node (level 0), we have $i_0 = i$ and $j_0 = j$ since $B_{0,0}[i..j]$ stores the first MSB of all the starting positions in T matching P, and it is easy to check that k_0 is equal to k.

More precisely, to answer *Select*$(P, 0, k)$, we first set $i_0 = i$, $j_0 = j$ and $k_0 = k$. We obtain the first MSB of p_k by comparing the number of 0's in $B_{0,0}[i_0..j_0]$ with k_0. Then, we iteratively apply the following step to retrieve the remaining bits of p_k. Suppose that after visiting the level-x node v of the binary tree so that v is augmented with some bit-vector $B_{x,y}$, we obtain i_x, j_x, k_x, and the $(x+1)$th MSB of p_k. To obtain the $(x+2)$th MSB, there are two cases:

- **(Case 1.)** If the $(x+1)$th MSB is 0, let i' and j' denote the number of 0's in $B_{x,y}[0..i_x - 1]$ and $B_{x,y}[0..j_x]$, respectively. Now, set $i_{x+1} = i' + 1$, $j_{x+1} = j'$ and $k_{x+1} = k_x$.

  Then, we follow the left child of the node v and obtain $B_{x+1,2y}$. We count the number of 0's in $B_{x+1,2y}[i_{x+1}..j_{x+1}]$. If the number is at least k_{x+1}, we report the $(x+2)$th MSB of p_k to be 0. Otherwise, we report 1.

- **(Case 2.)** If the $(x+1)$th MSB is 1, let r be the number of 0's in $B_{x,y}[i_x..j_x]$. Let i'' and j'' denote the number of 1's in $B_{x,y}[0..i_x-1]$ and $B_{x,y}[0..j_x]$, respectively. Now, set $i_{x+1} = i'' + 1$, $j_{x+1} = j''$ and $k_{x+1} = k_x - r$.

  Then, we follow the right child of the node v and obtain $B_{x+1,2y+1}$. We count the number of 0's in $B_{x+1,2y+1}[i_{x+1}..j_{x+1}]$. If the number is at least k_{x+1}, we report the $(x+2)$th MSB of p_k to be 0. Otherwise, we report 1.

Traversing each level of the binary tree requires a constant number of bit_rank queries on the corresponding bit-vector. Thus, each level takes $O(1)$ time to traverse, and the total time to answer the desired query is $O(\log n)$.

Range_Count$(P, 0, q)$ Query—the restricted Range_Count: To find the number of occurrences of P in T whose starting position is at most q, we will traverse the binary tree of our auxiliary data structure from the root downwards (based on the binary representation of q) so that at each node we visit, we update the number of desired occurrences of P (i.e., whose starting position is at most q) inferred by that node. Precisely, when we are visiting a level-x node in the binary tree, we are going to count the number of desired occurrences whose starting position shares exactly the first x bits with the binary representation of q, based on the $(x+1)$th MSB of q. For instance, at the root node (level 0), if the first MSB of q is 0, the number to be counted at this level is 0. Otherwise, if the first MSB of q is 1, the number to be counted is the number of 0's in $B_{0,0}[i..j]$. It is easy to verify that the required answer is the sum of number of desired occurrences reported at each level.

When we are visiting a level-x node, we identify the range i_x and j_x within the augmented bit-vector $B_{x,y}$ that stores all candidates of the desired occurrences of P which has not yet been counted. To answer *Range_Count*$(P, 0, q)$, we first set $i_0 = i$ and $j_0 = j$. Then, we iteratively apply the following step to find the number of desired occurrences of P. Suppose that after visiting the level-x node v of the binary tree so that v is augmented with some bit-vector $B_{x,y}$, we obtain i_x and j_x. To obtain the number of desired occurrences to be counted at level x, say occ_x, and to traverse to the next node at level $x+1$, there are two cases:

- **(Case 1.)** If the $(x+1)$th MSB of q is 0, we report $occ_x = 0$. Let i' and j' denote the number of 0's in $B_{x,y}[0..i_x - 1]$ and $B_{x,y}[0..j_x]$, respectively. Now, set $i_{x+1} = i' + 1$ and $j_{x+1} = j'$. In case $i_{x+1} > j_{x+1}$, we stop the traversal (This indicates that no desired occurrences has starting position sharing with binary representation of q for more than x bits). Otherwise, the next node is the left child of the node v with the augmented bit-vector $B_{x+1,2y}$.

- **(Case 2.)** If the $(x+1)$th MSB of q is 1, we report occ_x to be the number of 0's in $B_{x,y}[i_x..j_x]$. Let i'' and j'' denote the number of 1's in $B_{x,y}[0..i_x - 1]$ and $B_{x,y}[0..j_x]$, respectively. Now, set $i_{x+1} = i'' + 1$ and $j_{x+1} = j''$. In case $i_{x+1} > j_{x+1}$, we stop the traversal. Otherwise, the next node is the right child of the node v with the augmented bit-vector $B_{x+1,2y+1}$.

Traversing each level of the binary tree requires a constant number of bit_rank queries on the corresponding bit-vector. Thus, each level takes $O(1)$ time to traverse, and the total time to answer the desired query is $O(\log n)$.

Given the restricted version of the *Select* and *Range_Count* queries, we are ready to show how the general queries are performed.

Select(P, p, k) Query—the general Select: We first get the number of occurrences of P whose starting position in T is less than p. This can be done by calling *Range_Count*$(P, 0, p-1)$. Let r be such number. Then, we use *Select*$(P, 0, r+k)$ to obtain the desired answer. The total time is $O(\log n)$.

Range_Count(P, p, q) Query—the general Range_Count: This is done by computing *Range_Count*$(P, 0, p-1)$ and *Range_Count*$(P, 0, q)$. Their difference is the desired answer, and the total time required is $O(\log n)$.

Range_Report(P, p, q) Query: We first get the number of occurrence of P whose starting position in T is less than p. This can be done by calling *Range_Count*$(P, 0, p-1)$. Let r be such number. Then, we use *Select* query to report the $(r+1)$th occurrence, $(r+2)$th occurrence and so on of P in T until the position of the occurrence is greater than q. It is easy to verify that the occurrences reported are the desired answers. The total time is $O((1+occ)\log n)$ where *occ* is the number of occurrences of P with starting position between p and q in T.

Space Analysis: The space can be divided into two parts: (i) The binary tree, and (ii) the bit-vectors $B_{x,y}$. For the binary tree, there are $\Theta(n)$ nodes, and each node requires $\Theta(\log n)$ bits for storing the pointers to maintain the tree structure and to access the corresponding bit-vector $B_{x,y}$, so the total space for this part is $\Theta(n \log n)$ bits. To count the space incurred by the bit-vectors, we notice that for the bit-vectors augmented to nodes at the same level in the binary tree, each bit corresponds to distinct entry in the suffix array, so that the total number of bits for these bit-vectors is n. This implies that the space for the bit-vectors and the corresponding data structure of *bit_rank* query is bounded by $n + o(n)$ bits, and the total is bounded by $\lceil \log n \rceil \times (n + o(n))$ bits, which is $\Theta(n \log n)$ bits. Thus, the total space required by our data structure is $\Theta(n \log n)$ bits.

In summary, we have the following theorem.

Theorem 1 (Ordered Pattern Matching Structure) *Given a text $T[0..n-1]$, we can maintain a data structure in $\Theta(n \log n)$ bits such that for any pattern P, it supports Select and Range_Count queries in $O(|P| + \log n)$ time and Range_Report query in $O(|P| + occ \log n)$ time.*

If more space are allowed, we can further speed up the *Range_Report* query, and make it become $O(|P| + \log n + occ)$ time. The idea is that, along with each bit-vector $B_{x,y}$, we store the corresponding SA entries together with it. Then, we perform a search on the binary tree similar to that of *Range_Count*, and for each node we visit, we output (instead of count) the desired occurrences of that level. The total space is $\Theta(n \log^2 n)$ bits, or $\Theta(n \log n)$ words.

4 Applications

In this section, we describe some of the applications where our ranked/ordered pattern matching queries are useful. This is in no way an exhaustive list and we believe these primitives could have a lot more general applicability.

Aligned Pattern Matching. Consider two text arrays T_1 and T_2, both of size n characters. Imagine both arrays are written horizontally and T_1 is placed below T_2. As a query, we are given two patterns P_1 and P_2. Our task is to find all the text positions p such that P_1 matches with $T_1[p..]$ and P_2 matches at $T_2[p..]$. Let L_1 be the list of positions where P_1 matches T_1 and let L_2 be the list of positions where P_2 matches T_2. Then, our output is $L_1 \cap L_2$.

Instead of naively accessing each element in L_1 and L_2, we can apply Demaine et al.'s fast merging techniques [8] on L_1 and L_2 to compute the intersection. We use time within a multiplicative $\log n$ factor of theirs as we can access $L_1[k]$ and $L_2[k]$ in $\log n$ time. This problem can be extended directly to two-dimensional pattern matching in general where a rectangular text pattern is matched in a given two-dimensional text array.

Document Retrieval Problem. Consider the document retrieval problem of [19]. Here, the input text T comes as a collection of documents. Each document is a string of characters. Given a pattern P, the task is to report the list of documents which contain this pattern. Note that total number of occurrences of pattern P over the entire collection may far exceed total number of documents in which P occurs. We can solve this problem using our data structure.

Let the text T consist of concatenation of all documents, and let A be an array storing the starting positions of each document in text T. Now we report the documents one after the other. Let the current document being reported be d_i, and let p_i be the position of the last character of d_i in T. Then, $Select(P, p_i + 1, 1)$ will return the position of the next match. Now, we consult A to find which document this corresponds to, output that document and proceed to find the next document. If occ is the number of documents containing P, this takes $O(occ \log n)$ time.

Proximity Search. Given a text T and patterns P_1 and P_2, the task is to find all pairs of occurrences of P_1 and P_2 in T whose starting positions are within a distance d from each other. Let s be number of occurrences of P_1 and t be that of P_2 in T and $s \ll t$. Then, we can take the list of positions of P_1 and for each position p, run $Range\_Report(P_2, p - d, p + d)$ query. The total time required is $O((s + occ) \log n)$.

5 Working in External Memory

In the context of external memory data structure, our target is to minimize the number of I/Os involved to answer each query. Ferragina and Grossi [11] proposed the external memory version of a suffix array of T, which takes $O(n/b)$ pages and it can support (among other queries) finding the range in SA that matches a pattern P in $O(|P|/b + \log_b n)$ I/Os, where b is the number of words in a disk page.

To further support ordered pattern matching queries, we can transform our auxiliary data structure to make it work well in external memory. The basic idea is instead of having a binary tree augmented with bit-vector, we use a b-ary tree augmented with vectors of n characters, each taking $\log b$ bits. The total space is $O(n/b)$ pages, and it supports the *Select* and *Range_Count* queries in $O(\log_b n)$ I/Os, and supports the *Range_Report* query in $O(occ \log_b n)$ I/Os. We defer the details in the full paper.

6 Conclusion

We have proposed three ordered pattern matching queries for full-text retrieval systems. These queries can be useful in answering complex text queries efficiently, including document retrieval, proximity search and aligned pattern matching. While these queries can be considered as a two-dimensional range query and there are known geometric data structures supporting them quickly, we have provided an alternative solution using binary search tree augmented with bit dictionaries. This may be a crucial step in obtaining a compressed index for ordered pattern matching query.

References

[1] P. K. Agarwal and J. Erickson. Geometric Range Searching and Its Relatives. *Advances in Discrete and Computational Geometry*, 23:1–56, 1999.

[2] L. Arge, G. S. Brodal, R. Fagerberg and M. Laustsen. Cache-oblivious planar orthogonal range searching and counting, In *Proceedings of SoCG*, pages 160–169, 2005.

[3] L. Arge, V. Samoladas and J. S. Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing, In *Proceedings of PODS*, pages 346–357, 1999.

[4] R. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*, Addison Wesley, 1999.

[5] B. Chazelle. Lower Bounds for Orthogonal Range Searching, I: The reporting case. *Journal of the ACM*, 37:200–212, 1990.

[6] R. Cole, L.-A. Gottlieb and M. Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proceedings of STOC*, pages 91–100, 2004.

[7] O. de Krester and A. Moffat. Effective Document Presentation with a Locality-Based Similarity Heuristic. In *Proceedings of SIGIR*, pages 113–120, 1999.

[8] E. D. Demaine, A. Lopez-Ortiz and J. I. Munro. Adaptive Set Intersections, Unions, Differences. In *Proceedings of SODA*, pages 743–752, 2000.

[9] E. D. Demaine, A. Lopez-Ortiz and J. I. Munro. Experiments on Adaptive Set Intersections for text Retrieval Systems. In *Proceedings of ALENEX*, pages 91–104, 2001.

[10] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of FOCS*, pages 137–143, 1997.

[11] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Search in External Memory and Its Application. *Journal of the ACM*, 46(2):236–280, 1999.

[12] P. Ferragina, N. Koudas, S. Muthukrishnan and D. Srivastava. Two Dimensional Substring Indexing. In *Proceedings of PODS*, 2001.

[13] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[14] R. Grossi, A. Gupta and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of SODA*, pages 841–850, 2003.

[15] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, pages 397–406, 2000.

[16] G. Jacobson. Space-Efficient Static Trees and Graphs. In *Proceedings of Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[17] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[18] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[19] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of SODA* pages 657–666, 2002.

[20] P. Weiner. Linear Pattern Matching Algorithm. In *Proceedings of SWAT*, pages 1–11, 1973.